(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
25 April 2002 (25.04.2002)

PCT

(10) International Publication Number
WO 02/33556 A2

(51) International Patent Classification[7]: G06F 13/00

(21) International Application Number: PCT/US01/29850

(22) International Filing Date:
24 September 2001 (24.09.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/692,304 19 October 2000 (19.10.2000) US

(71) Applicant: **SUN MICROSYSTEMS, INC.** [US/US]; 901 San Antonio Road, MS PAL01-52, Palo Alto, CA 94303 (US).

(72) Inventors: **YANG, Liuxi**; 1260 Heather Stone Way, Sunnyvale, CA 94087 (US). **PHAM, Tung**; 243 Castillow Way, San Jose, CA 95119 (US).

(74) Agents: **ROSENTHAL, Alan, D.** et al.; Rosenthal and Osha L.L.P., Suite 2800, 1221 McKinney, Houston, TX 77010 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

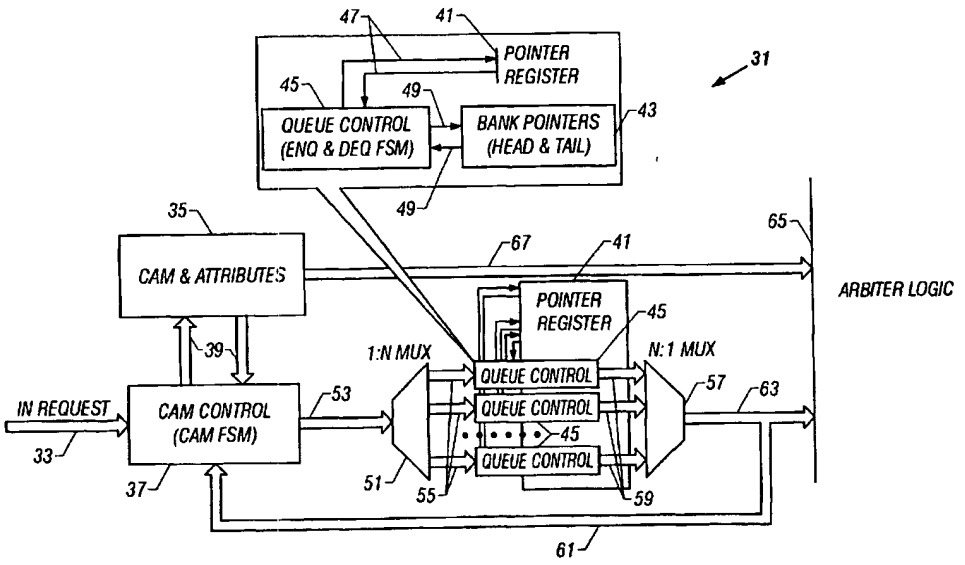
Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii)) for all designations*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii)) for all designations*

[Continued on next page]

(54) Title: DYNAMIC QUEUING STRUCTURE FOR A MEMORY CONTROLLER



(57) Abstract: A memory management system is disclosed including a processor, a memory control unit coupled to the processor, and a memory coupled to the memory control unit. The memory is configured in a plurality of banks, where the memory control unit includes a dynamic queuing structure, a pointer register defining a plurality of queues associated with the plurality of banks of the memory, an attributes register configured to store attributes of memory service instructions, a content addressable memory configured to store memory access addresses of the memory service instructions, and a queue control configured to control placement of memory service instructions in the plurality of queues based upon the attributes and the memory access address thereof.

WO 02/33556 A2

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

DYNAMIC QUEUING STRUCTURE FOR A MEMORY CONTROLLER

FIELD OF THE INVENTION

The present invention relates in general to computer systems and memory systems and in particular to hardware design that optimizes performance of an on-chip memory controller.

BACKGROUND OF THE INVENTION

Figure 1 shows an exemplary prior art multiprocessor environment 3. The multiprocessor environment 3 contains a plurality of processors 5. Each processor 5 is provided with an on-chip Memory Controller Unit (MCU) 7 that communicates with an associated off-chip memory 9. The memory 9 may, for example, be a synchronous dynamic random access memory (SDRAM). Each processor 5 communicates with its associated memory 9 under the control of MCU 7 via a memory interface 11. Each processor 5 also communicates with a system interface 13 via a system interface bus 15. Although three processors 5 are shown in Figure 1 for purposes of this exemplary discussion, the term "multiprocessor environment" as used in this application is intended to mean any environment that has more than one processor.

Functional block 17 is a schematic functional diagram of the workings of the MCU 7. The MCU 7 is responsible for initiating and controlling read, write, and associated memory requests with associated memory 9. Table 1 below lists an example of four types of incoming memory requests. Specifically, these are "MEM_READ", which is a read request to memory; "MEM_WRITE", which is a write request to memory; "MEM_CANCEL", which is a cancellation request to a previous read or write request; and "WRITE_RDY", which is a request indicating that write data is ready. When the MCU initiates one of these incoming memory requests, the request appears at functional location 19 in box 17 of the Figure 1. The request is then placed

in appropriate one of a plurality of queues 21. Each queue 21 corresponds with an associated bank (not shown) in memory 9. Once the incoming memory request is placed in the appropriate queue 21, it then waits for execution under control of the request arbiter 23. The request arbiter 23 monitors the memory requests resident in all of the request queues 21 and determines the order in which the requests should be carried out based upon any interdependencies that may exist between requests resident in the queues and based upon other control factors that are well known in the art.

Requests	Description
MEM_READ	A read request to memory; associated with id, bank id and address
MEM_WRITE	A write request to memory; associated with id, bank id and address
MEM_CANCEL	A cancellation to a previous read or write; associated with id
WRITE_RDY	Write data is ready, associated with id

TABLE 1. Incoming Memory Requests

Figure 2a illustrates the functional process that must be followed upon issuance of a read request. A requestor issues a read request at point A. The requestor may be, for example, software running on the system that makes the request via the system interface 13. The request is then sent to the MCU at point B. The MCU locates the requested data in associated memory 9 and, at point C, sends the located data to the requestor at point D.

Figure 2b illustrates the functional process that is followed for a write request. At point E, the requestor makes a request to write data, and this request is sent to the MCU at point F via system interface 13. At point F, the MCU must determine whether there is sufficient space in associated memory 9 to store the requested data. If the MCU can find available storage in memory 9, at point G the MCU generates a target ID which is sent to the requestor at point H. Upon receipt of the target ID, the requestor sends the data at point I to the MCU at point J, which is then written to the allocated location in memory at point K. It should be apparent that, between points F and G, if the MCU can

not find or currently does not have sufficient memory space to allocate the write request, the write operation stalls until the MCU is able to find sufficient storage space. Accordingly, the write operation inherently takes longer to execute then a read operation, and may in fact take quite a long time if the necessary storage space is not immediately available in the associated memory 9.

Returning now to Figure 1, it is apparent that when a sequence of read and write requests are placed in the request queues 21, if a single write request is at the front of the queue and is taking a long time to execute, the queue may back up with multiple other read and write requests which, if they where at the front of the queue, could be executed more quickly. Therefore, the MCU queue structure shown in prior art Figure 1 may suffer from lack of optimization in handling read and write requests. Ultimately, because the queues 21 are of finite size, if one queue becomes backed up, this will cause a halt in operation of system interface 13, in turn diminishing performance of the entire multiprocessor system 3.

SUMMARY OF THE INVENTION

In one aspect, the invention comprises a memory controller adapted to receive memory service instructions from a processor and adapted to communicate with an external memory organized in a plurality of banks. In one embodiment, the memory controller comprises a plurality of request queues corresponding respectively to the plurality of banks, means for monitoring status of the plurality of queues, means for discriminating characteristics of incoming memory service instructions, and means for allocating the incoming memory service instructions to the plurality of queues based on the status of the queues and the characteristics of the memory service instructions.

In another aspect, the invention comprises a memory management system comprising a processor, a memory control unit coupled to the processor, and a memory coupled to the memory control unit, where the memory may be configured in a plurality of banks. In one embodiment, the memory control unit includes a dynamic queuing structure comprising a pointer register defining a plurality of queues associated with the plurality of banks of the memory, an attributes register configured to store attributes of memory service instructions, a content addressable memory configured to store memory access addresses of the memory service instructions, and a queue control configured to control placement of memory service instructions in the plurality of queues based upon the attributes and the memory access address thereof.

In another aspect, the invention comprises a method of controlling memory service instructions in a computer system having a processor, a memory configured in a plurality of banks and a memory control unit defining a plurality of queues corresponding to the plurality of banks. In some embodiments the method comprises monitoring a status of the plurality of queues, discriminating characteristics of an incoming memory service instruction, and placing the incoming memory service instruction in one of the plurality of queues
based upon the characteristics thereof and the status of the queues.

Other aspects and advantages of the invention will be apparent from the following description and the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows a prior art memory controller in a multiprocessor environment.

Figure 2a is a description of memory read protocol sequences.

Figure 2b is a description of memory write protocol sequences.

Figure 3 is a block diagram of a dynamic queuing structure in accordance with an embodiment of the invention.

Figure 4 illustrates the storage used by a dynamic queuing structure in accordance with an embodiment of the invention.

Figure 5 shows the inputs and outputs of a Content Addressable Memory (CAM) in accordance with an embodiment of the invention.

Figure 6 is a timing diagram describing CAM read blocking in accordance with an embodiment of the invention.

Figure 7 is a flow chart describing read/write request processing in accordance with an embodiment of the invention.

Figure 8 is a description of a CAM update Finite State Machine in accordance with an embodiment of the invention.

Figure 9 is a schematic description of how the queuing process is done and how memory requests are linked into a queue in the Memory Control Unit in accordance with an embodiment of the invention.

Figure 10 is a state transition diagram of an enqueue and dequeue Finite State Machine in accordance with an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention provides a system and method by which handling of memory requests within the MCU may be optimized in such a way to prevent unnecessary degradation of system performance due to memory request backlog in the request queues. As described with reference to the prior art, a backlog in a single queue may result in degradation of system performance even if other queues are running near empty. Moreover, a single write request at the front of a queue that takes a long time to execute can create a great backlog of other read and write requests that, were it not for the slower write request, could be executed more quickly. In accordance with embodiments of the invention, a memory request that is faster to execute but arrives later in the MCU can be inspected to ascertain whether its execution is dependent on any other instructions that arrived earlier in the MCU. If no such

dependency exists, the later arrived memory instruction can be placed ahead of the slower one. Embodiments of the invention incorporate this capability in the MCU dynamic queuing structure for overall increase in the processor speed.

In modern day computers, memory units such as Synchronous Dynamic Random Access Memory (SDRAM) are organized in banks. Each bank, with its distinct identity tag, is accessible from outside by the MCU through a control and data bus. Associated with each data bank is a finite-sized queue in the MCU, capable of holding memory request instructions for that queue. During computer operations, the number of memory requests (*e.g.*, Read or Write) arriving in the queues for different banks will be different. Some queues will be more full than others. It is likely that the queue for one memory bank may become full, which may lead to a system halt, while queues for other banks will have more empty spaces. Therefore, incorporation of the capability by which unused queuing space of one bank could be temporarily allocated to the needy one and taken back when not used leads to a more efficient use of the finite storage space available in the MCU. This results in an overall increase in computer speed. Embodiments of the present invention incorporate this capability in the MCU.

Memory requests (service instructions) arriving at the MCU may be processed to see whether instructions destined for the queue for a particular memory bank could be placed ahead of other instructions already in the queue. Different memory service instructions are of different time lengths in units of processor clock cycles and are subdivided into several executable pieces for each processor clock cycle. In accordance with embodiments of the invention, the MCU may be designed so that, during a single processor clock cycle, multiple memory service instructions can be in different stages of execution toward a designated goal. In the art of computer instruction execution, this is known as the pipelining process. Embodiments of the present invention

incorporate this process in the MCU to speed up the formation of the dynamic queuing structure.

Specific exemplary embodiments of the invention will now be described with reference to the accompanying figures. Like elements in like figures are denoted by like reference numerals for clarity.

Shown in Figure 3 is a block diagram of a dynamic queuing structure **31** of an MCU in accordance with one embodiment of the invention. The MCU is resident in a microprocessor chip in this example, but could also be extended. The MCU may function in a single processor environment as well as in a multi-processor environment. The MCU receives memory requests at **33** from a System Interface Unit (SIU) also typically resident in the microprocessor chip (not shown here). The dynamic queuing structure **31** is implemented with a Content Addressable Memory (CAM) and attributes unit **35** and a CAM Control Finite State Machine (CAM FSM) **37**. The CAM & Attributes unit **35**, which is a storage unit in the MCU, records the memory access addresses and detects dependencies for the incoming requests. The function of this unit **35** is further explained with the help of Figures 4, 5, 6 and Tables 2 and 3 below. When incoming memory requests **33** are received, the CAM FSM **37** updates the CAM & Attributes unit **35** via communication paths **39**. The operation of the CAM FSM is explained in detail below with reference to Figures 7 and 8.

An outside SDRAM that the MCU will access has its memory organized in banks. Each bank is identifiable with an identity tag (ID). Corresponding to each memory bank of the SDRAM there is a request queue in the MCU. In Figure 3, the pointer registers **41** together with the head and tail pointers of the bank pointers **43** are used to form the request queues. The pointer registers **41** are formed, for example, out of a Random Access Memory (RAM) storage in the MCU. The pointer registers **41** and head and tail pointers **43** are further discussed with reference to Figure 4 below. Formation of the request queues is explained below with reference to Figure 9.

Corresponding to each request queue there is a Queue Control Finite State Machine (QC FSM) **45** that controls the formation and reordering of the queues.

Within the MCU, the memory space allocated to each request queue is not fixed but allocated, as needed, by the QC FSM **45**. Formation and reordering of the request queues for each bank of the SDRAM are controlled by the QC FSM **45** via paths **47** for the pointer registers and paths **49** for the bank pointers. Operation of the QC FSM **45** is further explained below with the aid of Figure 10 and Table 5.

The CAM FSM **37** controls formation and reordering of the request queues by providing inputs to QC FSM **45** through a 1:N demultiplexer (DEMUX) **51.** N is the total number of request queues, which is same as the total number of SDRAM banks. The DEMUX **51** is connected to the CAM FSM **37** via incoming path **53** and to the QC FSM **45** via outgoing paths **55**.

Current status information about all the queues is provided to the CAM FSM **37** through a N:1 multiplexer (MUX) **57**. The incoming paths 59 to the MUX **57** lead to the outgoing path **61** to the CAM FSM **37.** Another outgoing path **63** goes to an arbiter logic unit **65** of the MCU. The CAM & Attributes unit **35** provides updated inputs to the MCU arbiter logic unit **65** via path **67.** The arbiter logic unit **65** controls a Request Dispatch Buffer (not shown in Figure 3) where a memory request's Row Access Strobe (RAS) and Column Access Strobe (CAS) are formed. Under the control of the arbiter logic unit **65,** the RAS and CAS are generated and scheduled in the Request Scheduler of the MCU.

As previously explained, Table 1 above lists the incoming memory requests that are processed by a memory controller. MEM_READ/MEM_WRITE is a memory read/write request. Each memory read/write request includes an identity tag (ID); an index of the entry in MCU storage; a bank ID; a queue ID in the MCU; and the memory address to the

bank indicated by the bank ID. If the memory block is unavailable, *e.g.*, currently owned by another device's cache, that device will signal the MCU to cancel the memory request. In such a case, the MCU receives a MEM_CANCEL request. The MEM_CANCEL request has an ID associated with it which is the same index ID used by MEM_READ/MEM_WRITE. Thus, the MEM_READ/MEM_WRITE can be easily cancelled.

A write request does not have data coming with its transactions because a space for the data has to be found first. The original device will send the data packet once the target ID is received. WRITE_RDY indicates data has been received after the target ID is issued. WRITE_RDY also has an ID associated with it. The ID points to the same ID that the MEM_WRITE is associated with. Generally, the processor clock is running at a much higher speed than the system interface clock. Because all memory requests are coming from the system interface, the rate at which request arrive is only a fraction of the processor clock rate. In the MCU design discussed here, it is assumed that, at most, the MCU receives a MEM_READ/MEM_WRITE every four processor cycles. Every four consecutive processor cycles can be grouped together. In the present discussion, it is assumed that all MEM_READs/MEM_WRITEs can only come at the first cycle of each group.

Figure 4 is a schematic depiction of the storage layout **71** used by a dynamic queuing data structure in accordance with one embodiment of the invention. The transaction information and queuing order are stored and manipulated in these stages. The address storage space **73** is for the Content Addressable Memory (CAM) of the unit **35**. The CAM records the memory access addresses and detects dependencies for new incoming requests. The entries in CAM **73** are indexed by the incoming transaction ID. The ID range defines the MCU's processing capability for the pending requests. In addition to the address, the bank numbers of the SDRAM are also stored in the CAM **73** for comparison, because the address range of requests going into different

banks may overlap. The CAM **73** performs three operations: content update (wr); content read (rd); and compare (cmp). "wr" writes a new request's address and bank information into CAM. "rd" reads out the address and bank information of a request. "cmp" generates the entries that match the compared address and bank number. The match is indicated by a bit vector. To simplify the CAM design, at any given time, out of the above three operations, only one can be active.

The second major block in the storage layout of Figure 4 is the pointer storage **75**. The pointer storage **75** may be, for example, a typical Random Access Memory (RAM) array. Storage **75** has a read port and a write port. The read and write operations are not active at the same cycle.

Storage **81** in Figure 4 contains the attributes registers. The attributes in storage **81** are the information to be passed along to the next stage in the MCU. These registers also book-keep the status changes for each request so that appropriate operations can be performed for each request. These are important data structures that guarantee correct memory access. The storage **81** may be implemented, for example, with D flip flops. Representative attributes as shown in Figure 4 are: "valid" **83**; "Req_type" **85**; "Queued" **86**; "Rdy" **87**; "Cancelled" **88**; and "Bnk_num" **89**.

Each entry in the pointer storage **75** maps to one entry in the CAM **73** and in the attributes register storage **81**. Head **77** and tail **79** bank pointers correspond to each SDRAM bank and, together with pointer registers in storage **75** form the request queue for that bank. The head **77** and tail **79** bank pointers may be constructed, for example, using flip flops.

Figure 5 is a schematic diagram illustrating the CAM (storage **73**) inputs and outputs. The CAM structure is clocked with the clock input **92**. The three operational inputs to the CAM are content update (wr) **93**, content read (rd) **94**, and compare (cmp) **95**. The other inputs are transactions ID "indx" **96** and

data input "d_in" **97**. The output signals are data out "d_out" **98** and "cam_hit" **99**, indicating dependencies for the new incoming request.

The output results for each operation take effect at the next cycle after the operation input becomes active. By spacing the three operations into different cycles, the CAM structure design is greatly simplified. Although constructing the CAM with more ports will allow the MCU to take requests at a higher rate, such may not be necessary because the processor's clock is much higher than the system interface clock.

	Read	Write	Compare
rd	1	0	0
wr	0	1	0
cmp	0	0	1
indx	valid	valid	-
d_in	-	valid	valid
d_out	valid	-	-
cam_hit	-	-	valid

TABLE 2. CAM Truth Table

Table 2 shows a CAM Truth Table in accordance with an embodiment of the invention. This truth table indicates the logical conditions of the inputs and outputs of the CAM for the three operations (read, write and compare) performed during the processing of memory requests.

Operations	Description
rqst_wr	a new request comes in
rqst_cncl	a request is canceled
rqst_wrdy	a write request's data is ready
rqst_clr	MCU slef generated, clear the request after it has been processed
rqst_que	queue the request in

TABLE 3. Attribute Update Operations

Table 3 describes updating operations of the attributes storage **81** (Figure 4). The update operations shown are generated by execution of memory

service requests in a manner described below with reference to Table 4. At any given processor clock cycle, only one of "rqst_wr", "rqst_cncl", "rqst_wrdy", "rqst_clr", and "rqst_que" can be active.

Figure 6 shows an exemplary timing diagram for CAM read scheduling. Shown here is a processor clock signal **133**, a new read request signal **135** and a CAM Read Block signal **137**. To simplify the CAM design, no two of CAM read, CAM write and CAM compare are allowed to happen the same cycle. To prevent a CAM read conflict with the CAM write and CAM compare operations, any CAM read is blocked for two clock cycles if another read or write request is received. As shown in the diagram, once a new request **135** comes in, there will be no CAM reads for next two clock cycles. Blocking CAM reads does not impact performance because a SDRAM cycle is much larger than four processor cycles. Most importantly, without the above scheduling, the CAM has to be multi-ported, thus increasing cost.

Cycle	1	2	3	4	5
MEM_READ (hit)	valid	cmp	cam_wr,rqst_wr	enq	enq
MEM_WRITE (miss)	valid	cmp	cam_wr,rqst_wr	enq	-
MEM_READ (hit)	valid	cmp	cam_wr, rqst_wr	enq	-
MEM_WRITE (miss)	valid	cmp	-	-	-
MEM_READ (q empty)	valid	cam_wr, rqst_wr, enq	-	-	-
MEM_WRITE (q empty)	valid	cam_wr	-	-	-
MEM_CANCEL	valid	att update	-	-	-
WRITE_RDY	-	valid	enq	-	-

TABLE 4. Request Process

Table 4 lists the processor cycle by processor cycle detailed breakdown of activities for all the memory input request operations. It includes all the cases describing how the queuing mechanism should work in processor clock time sequence. This is in fact a description of the pipelining process of memory request servicing by the MCU. The first row has the processor cycle numbers that a memory request servicing operation could be spreading across.

In this exemplary description, it takes up to 5 processor clock cycles to execute MEM_READ (hit), the longest request. The first through the fifth columns for each type of memory request list the various logic and control update steps to be executed during the processor clock period corresponding to that column.

Referring back to Figures 4 and 5, the MEM_READ (hit) request has five executable steps spread over five processor clock cycles. The first step updates the attribute status to be "valid" **83**, in the first cycle. The second step executes compare operation, "cmp", in the CAM **95** in the second cycle. The third executable steps are CAM update operation with "cam_wr" **93** and attributes update operation with "rqst_wr" (Table 3) in the third cycle. The fourth executable step is the attributes update "enq" operation **86** in the fourth cycle. The fourth executable step is repeated in the fifth cycle.

The next MEM_WRITE (miss) request has four executable steps spread over four processor clock cycles. The first step updates the attribute **81** status to be "valid" **83** in the first cycle. The second step executes compare operation, "cmp", in the CAM **95** in the second cycle. The third executable steps are CAM update operation with "cam_wr" **93** and attributes update operation with "rqst_wr" (Table 3) in the third cycle. The fourth executable step is the attributes update "enq" operation **86** in the fourth cycle.

The next MEM_READ (hit) request has four executable steps spread over four processor clock cycles. The first step updates the attribute **81** status to be "valid" **83** in the first cycle. The second step executes compare operation, "cmp", in the CAM **95** in the second cycle. The third executable steps are CAM update operation with "cam_wr" **93** and attributes update operation with "rqst_wr" (Table 3) in the third cycle. The fourth executable step is the attributes update "enq" operation **86** in the fourth cycle.

The next MEM_WRITE (miss) request has two executable steps spread over two processor clock cycles. The first step updates the attribute **81** status

to be "valid" **83** in the first cycle. The second step executes compare operation, "cmp", in the CAM **95** in the second cycle.

The next MEM_READ (q empty) request has two executable steps spread over two processor clock cycles. The first step updates the attribute **81** status to be "valid" **83** in the first cycle. The second executable steps are CAM update operation with "cam_wr" **93**, attributes update operation with "rqst_wr" (Table 3); and the attributes update "enq" operation **86** in the second cycle.

The next MEM_WRITE (q empty) request has two executable steps spread over two processor clock cycles. The first step updates the attribute **81** status to be "valid" **83** in the first cycle. The second executable step is the CAM update operation with "cam_wr" **93** in the second cycle.

The next MEM_CANCEL request has two executable steps spread over two processor clock cycles. The first step updates the attribute **81** status to be "valid" **83** in the first cycle. The second executable step is the attribute update operation **81** in the second cycle.

The next WRITE_RDY request has three executable steps spread over three processor clock cycles. The first step is no operation in the first cycle. The second step updates the attribute **81** status to be "valid" **83** in the second cycle. The third executable step is the attributes update "enq" operation **86** in the third cycle.

This methodology supports an incoming request ratio of 1 request every 4 clocks or more. Practically, the incoming rate of requests is above 1 every 5 processor cycles. Obviously, at any given cycle, there is at most one operation. A previous memory read/write request can be cancelled by a MEM_CANCEL request. This will result in a removal of a request from an ordered queue (dequeue). Since dequeue operation requires CAM read, whenever there is a new request, cycles 2, 3 are blocked from CAM read (Figure 6). In this way, every 4 processor cycles, there are at least 2 cycles for CAM read to issue a

request to SDRAM. How the requests in the CAM are dispatched and scheduled is well known and thus is not included in this description.

Figure 7 is an exemplary flow chart of read/write request processing in accordance with an embodiment of the invention. This explains the basis of designing the CAM FSM **37**. The basic idea is to let a write request "float" until its data is ready or if there is dependency. The flowchart shows that a read will always get into its queue if its queue has no write, and that the queue empty status can always be obtained in advance.

When a new request (**ST3**) comes in to the MCU through the SIU, it is given a check (**ST5**) for Read request. If the answer is yes, then a further check is given for write pending (**ST7**). If write pending is yes, then the new request enters the CAM Write float state (**ST9**). If there is no write pending, a check for dependency, CAM Compare (**ST11**) is given. During the CAM Compare the new request stays in the CAM Write float state (**ST9**). If there is a hit (**ST13**), meaning a read hits a write, then the write gets into the queue first (**ST15**). Then the read itself enters the queue (**ST17**). If there is not a hit, the read gets into the queue (**ST17**).

If the new request is not a Read request, then a check (**ST19**) is given for Write Pending. If there is a write pending, then the new request enters the CAM Write float state (**ST21**), which is the same as the (**ST9**) float state. Again, as stated earlier, the new request stays in the CAM write float state until the write data is ready or the CAM Compare determines existence of a dependency. The write checks dependency with the CAM. If it hits a Write (**ST25**), the latter will be queued (**ST27**) and the new Write stays in float. When a WRITE_RDY comes in, the write that WRITE_RDY points to will be queued. If there is not a hit, the Read gets into the queue (**ST29**) and the new Request process is completed (**ST31**). A write will stay in float at CAM Write (**ST21**) if its queue has no write.

When a MEM_CANCEL comes in, the cancelled attribute of the pointed request is set. The request will be dequeued silently at top of the queue.

In this implementation, one queue for each bank has been chosen. Multiple queues per bank may also be used, depending on design considerations.

Figure 8 shows an exemplary CAM update Finite State Machine (FSM) **141** that updates the CAM **35**. The CAM entries in **73** (Figure 4), Pointer Register entries in **75** (Figure 4), and Attribute entries in **81** (Figure 4) have exactly the same indexing mechanism. In other words, among the three entries, there exists a one-to-one mapping relationship. It is this relationship that facilitates assignment of an entry to a particular queue. The three states are CAM_IDLE **143**, CAM_CMP **145** and CAM_WRT **147**.

Until a new request comes in, the CAM stays in the CAM-IDLE state **143**. A new request with no write pending (new_rqst, no_write) **155** brings the CAM to CAM_CMP **145**, the CAM Compare state. Until the dependency is checked, the CAM makes the transition **153** into the CAM_WRT float state **147**. If the dependency check hits a write already received and in a float state, the latter is queued and the new write stays in float state **147**. On the other hand, a new request with a previous write pending (**149**) brings the CAM to the CAM_WRT float state **147**. After the last WRITE_RDY is serviced and there are no dependency check pending, the CAM makes the transition **151** to CAM_IDLE **143**.

Figure 9 shows how the queuing process **181** is done and how memory requests are linked into a queue in the MCU. Every memory bank has one request queue. The Heads **187** (or **195**) and Tails **189** (or **197**) together with the contents in the Pointer Register **185** (or **193**), form ordered queues. The pointer registers **185** and **193** are in the pointer storage RAM **75** of Figure 4. The first part **183** of Figure 9 shows a queue organization. When a new request comes in, say at cycle 1(Table 4), the CAM comparison "cmp" decides in

which bank's queue it will be queued in. Following the example in part **191** of the Figure, at cycle 2, entry 12 in address CAM will be entered with the address indexed by the request ID 12, while entry 9 (indicated by the tail register **189**) of the pointer register **185** will be updated with number 9. In the meantime, the tail register **197** is updated with 12. Therefore, before the new request comes in, this bank has a queue **199** that has CAM entries 1,3,6, and 9. After the new request comes in, the queue **199** has entries 1,3,6,9, and 12. Each queue is also associated with state bits to indicate the status of the queue, for example, an empty status. This is implemented in the FSM.

Figure 10 shows the Enqueue and Dequeue Finite State Machine **201** with the states and the associated state transitions. This FSM implements the reordering of the queues of memory service instructions for the memory banks. In this exemplary diagram, the FSM has four states: Q_EMPTY, **203**; Q_VALID, **205**; Q_SERVED, **207**; and Q_FETCHED, **209**.

State	Description
Q_EMPTY	no transaction in this queue
Q_SERVED	queue, not empty but no transaction at head
Q_FETCHED	the head request is being processed and send to SDRAM
Q_VALID	a valid transaction at head

TABLE 5.

Table 5 provides a description of these four states. Q_EMPTY describes no transaction in the queue. Q_VALID describes a valid transaction at head. Q_SERVED describes a queue not empty but no transaction at head. Q_FETCHED describes the head request is being processed and sent to SDRAM.

When the queue is empty (Q_EMPTY), to start a queue, both the head and the tail should be filled with the same valid ID pointing to a CAM and the same Pointer Register entry. When the head has a valid request (Q_VALID), it needs to participate in the request arbitration for getting the request served.

When the head of the queue is dispatched to the SDRAM(Q_SERVED), the next request should be fetched to the queue location. When the head is dequeued, the queue needs to move forward. It is unnecessary to have the capability to fetch a request to a queue just in one processor cycle because the SDRAM can only process one request in a few processor cycles. So, there is one extra slot (Q_FETCHED) to get the next request into the head location.

This state machine is passive to the new enqueuing request. A new request **215** will be queued into a queue corresponding to the target memory bank. Therefore, the FSM stays (**211**) in Q_EMPTY **203** or stays (**213**) in Q_VALID **205** until it is possible for this queue to have a chance to send its head to the SDRAM. This is reasonable since for every 4 processor cycles there are 2 cycles that the CAM is free for read operations. Once the head is dispatched to the SDRAM, the FSM will transition (rqst_dispatch & near_empty, **217**) to either the Q_EMPTY state **203** or transition (rqst_dispatch & ~near_empty, **219**) to the Q_SERVED state **207** for not near empty status of the queue. The "near empty" status can be easily obtained by a head-tail comparator not explicitly shown here. Q_EMPTY **203** means no more valid transactions are in the queue. The Q_SERVED state **207** means that the head is invalid and that the next one needs to be fetched in.

The "fetch" operation actually needs two steps. First, the content of the Pointer Register pointed by the head is read out (que_forward, **221**). Next, the content is written into the head (forward_done, **223**). Therefore, there is one extra state (Q_FETCHED, **209**) for scheduling this two-step operation. Before the forward_done operation **223** actually takes place, the FSM waits (**225**) in the Q_FETCHED state **209**. Again, since the SDRAM is slow when compared to the processor clock, two-step operation (**221** and **223**) does not affect the SDRAM performance. Indeed, this SDRAM bank might need to wait more , *e.g.*, a few tens of processor cycles, before it can serve another SDRAM request.

Referring back to Figure 3, the incoming memory requests (**33**) are processed into appropriate request queues by the dynamic queuing structure forming system described above. The request queues are connected to the Request Dispatch Buffer where the RAS (Row Access Strobe) and CAS (Column Access Strobe) are formed completely under the control of the Arbiter logic unit **65**. The RAS and CAS are then scheduled for SDRAM access under control of the Arbiter **65**.

While the invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.

CLAIMS

What is claimed is:

1. A memory controller adapted to receive memory service instructions from a processor and adapted to communicate with an external memory organized in a plurality of banks, the memory controller comprising:

a plurality of request queues corresponding respectively to the plurality of banks;

means for monitoring status of the plurality of queues;

means for discriminating characteristics of incoming memory service instructions; and

means for allocating the incoming memory service instructions to the plurality of queues based on the status of the queues and the characteristics of the memory service instructions.

2. The memory controller of claim 1, further comprising means for reordering memory service instructions within the plurality of queues.

3. The memory controller of claim 1, further comprising arbiter logic operatively coupled to the plurality of queues and configured to execute the memory service instructions.

4. The memory controller of claim 1, wherein the characteristics comprise type of memory service instruction.

5. The memory controller of claim 1, wherein the characteristics comprise interdependency with other memory service instructions.

6. The memory controller of claim 4, wherein the means for allocating places a memory service instruction in a queue depending upon the type of memory service instruction.

7. The memory controller of claim 5, wherein the means for allocating places a memory service instruction in a queue depending upon interdependency with other memory service instructions.

8. The memory controller of claim 1, wherein the means for allocating further comprises means for maintaining a memory service instruction in a float state pending resolution of another operation.

9. A memory management system, comprising:

a processor;

a memory control unit coupled to the processor; and

a memory coupled to the memory control unit, the memory being configured in a plurality of banks;

wherein the memory control unit includes a dynamic queuing structure comprising:

a pointer register defining a plurality of queues associated with the plurality of banks of the memory;

an attributes register configured to store attributes of memory service instructions;

a content addressable memory configured to store memory access addresses of the memory service instructions; and

a queue control configured to control placement of memory service instructions in the plurality of queues based upon the attributes and the memory access address thereof.

10. The memory management system of claim 9, wherein the memory control unit is integrated with the processor.

11. The memory management system of claim 9, further comprising memory service instruction execution arbiter logic.

12. A method of controlling memory service instructions in a computer system having a processor, a memory configured in a plurality of banks and a memory control unit defining a plurality of queues corresponding to the plurality of banks, the method comprising:

monitoring a status of the plurality of queues;

discriminating characteristics of an incoming memory service instruction; and

placing the incoming memory service instruction in one of the plurality of queues based upon the characteristics thereof and the status of the queues.

13. The method of claim 12, wherein the act of discriminating comprises determining a type of instruction.

14. The method of claim 12, wherein the act of discriminating comprises

determining interdependency with other memory service instructions.

15. The method of claim 12, wherein the act of monitoring comprises determining current capacity of the plurality of queues.

16. The method of claim 12, wherein the act of monitoring comprises determining characteristics of memory service instructions in the queues.

17. The method of claim 16, wherein the characteristics include type of memory service instruction.

18. The method of claim 16, wherein the characteristics include interdependency with other memory service instructions.

19. The method of claim 12, wherein the act of placing comprises holding the incoming memory instruction in a float state pending outcome of another operation.

20. The method of claim 12, further comprising optimizing queuing by reordering memory service instructions in the plurality of queues.

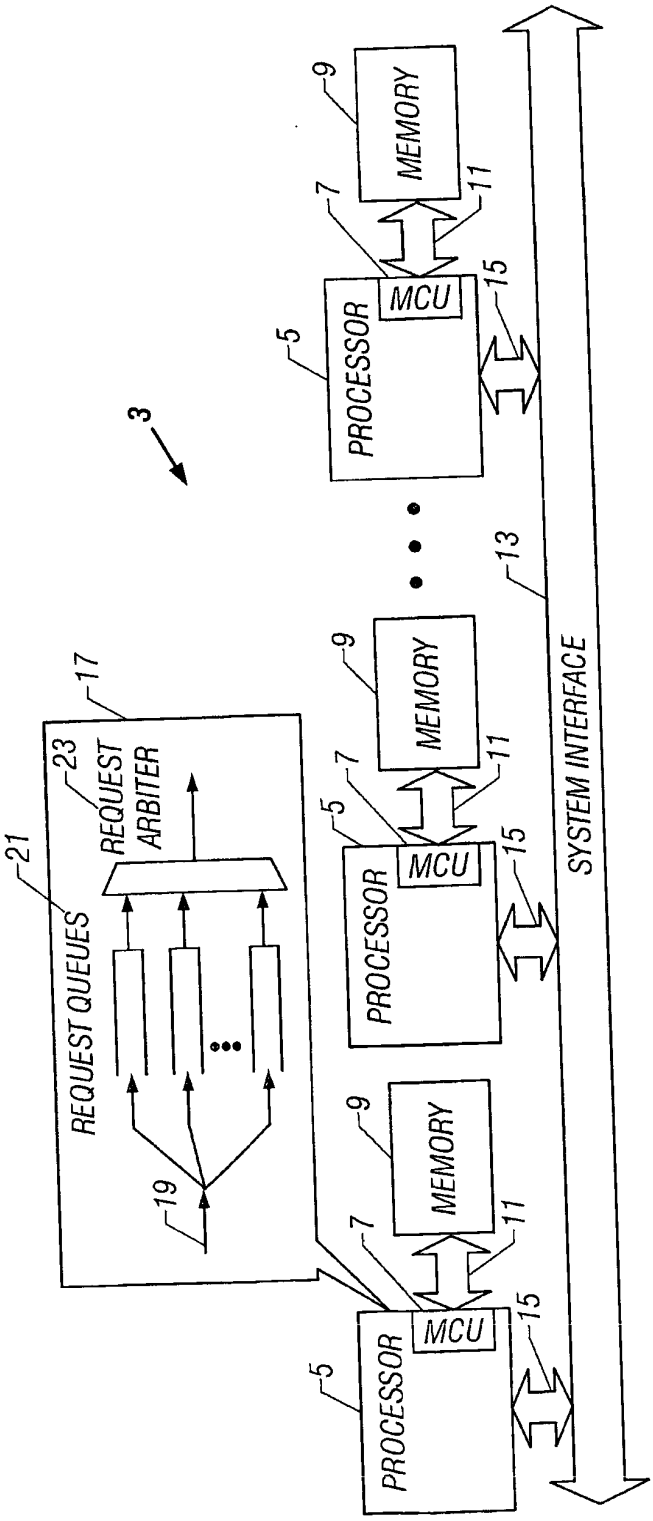


FIG. 1
(Prior Art)

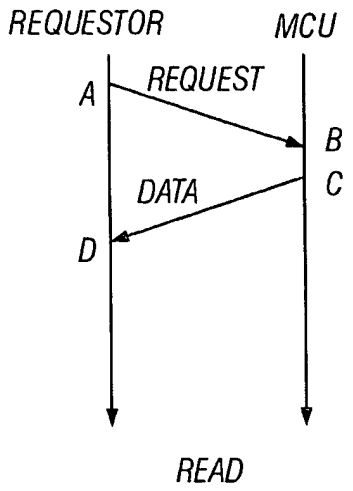


FIG. 2A
(Prior Art)

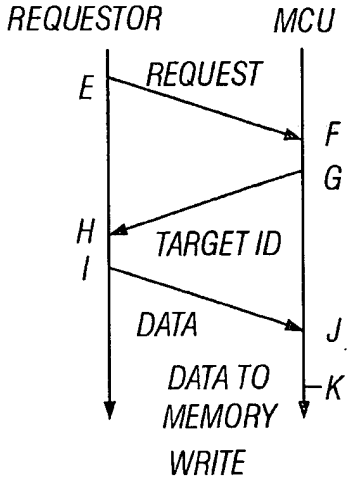


FIG. 2B
(Prior Art)

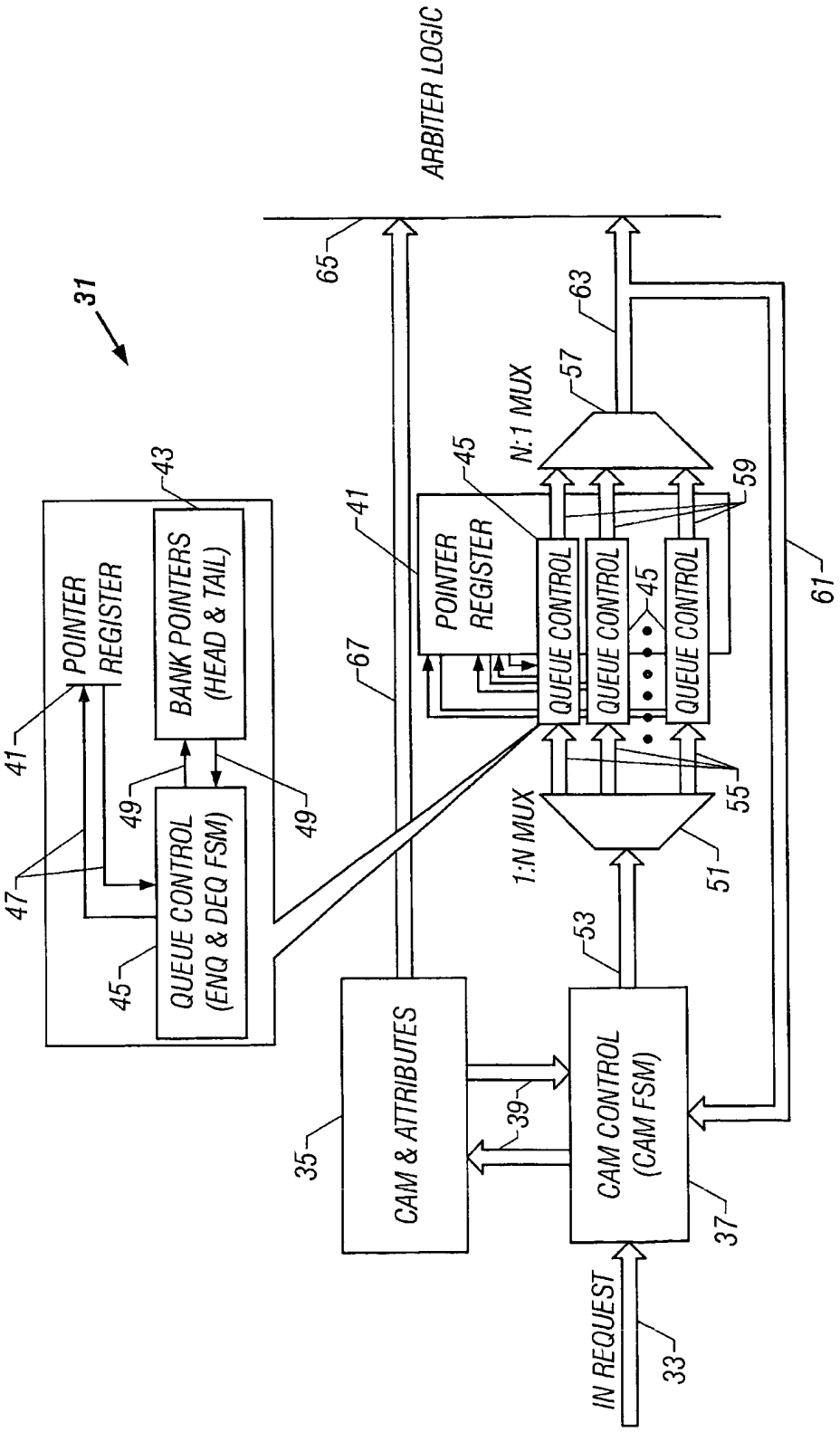

ARBITER LOGIC
65
31
63
57
N:1 MUX
45
59
43
41
POINTER REGISTER
BANK POINTERS (HEAD & TAIL)
QUEUE CONTROL
QUEUE CONTROL
QUEUE CONTROL
61
67
49
49
1:N MUX
55
51
47
QUEUE CONTROL (ENQ & DEQ FSM)
53
CAM & ATTRIBUTES
39
CAM CONTROL (CAM FSM)
35
37
IN REQUEST
33

FIG. 3

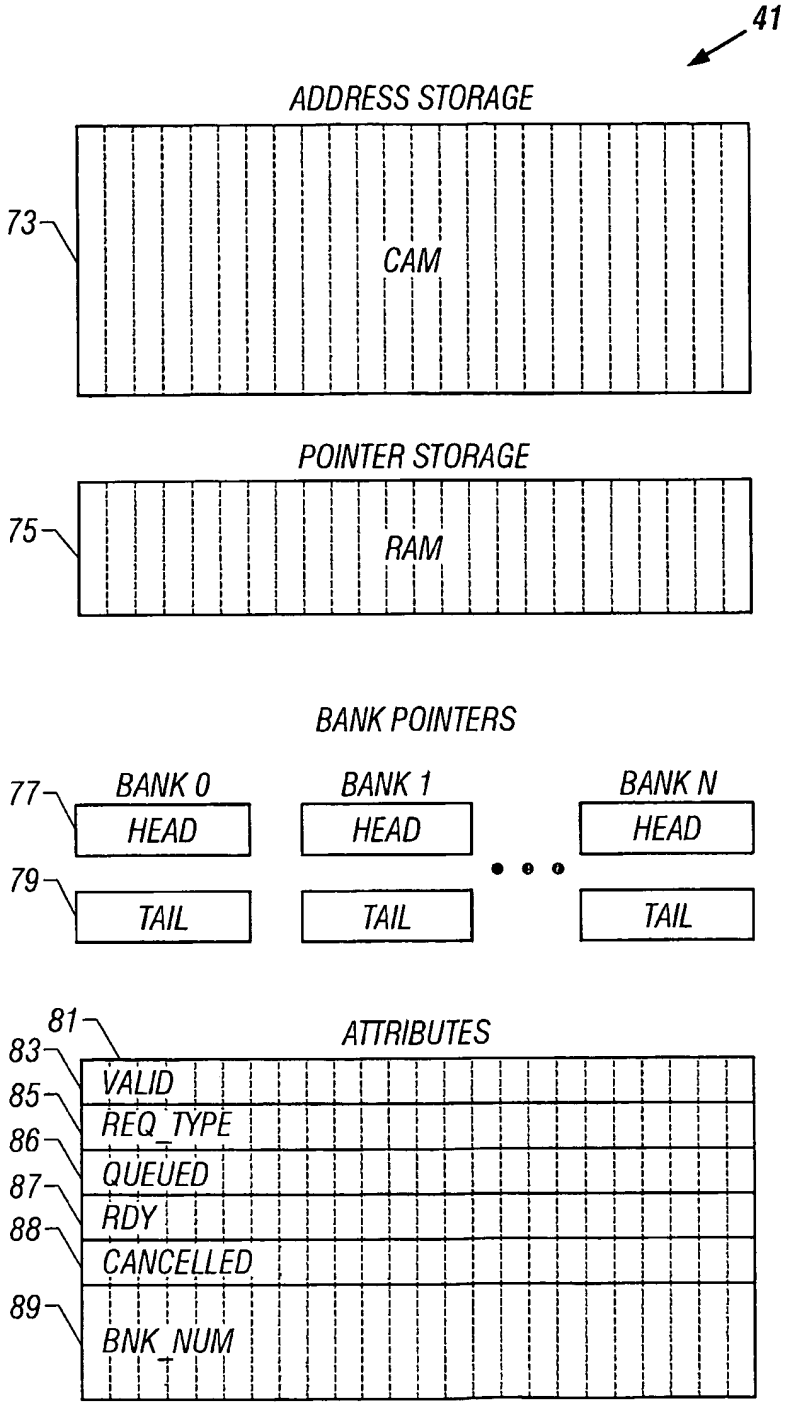

41
ADDRESS STORAGE
73
CAM
POINTER STORAGE
75
RAM
BANK POINTERS
77
BANK 0
BANK 1
BANK N
HEAD
HEAD
HEAD
79
TAIL
TAIL
TAIL
81
ATTRIBUTES
83
VALID
85
REQ_TYPE
86
QUEUED
87
RDY
88
CANCELLED
89
BNK_NUM

FIG. 4

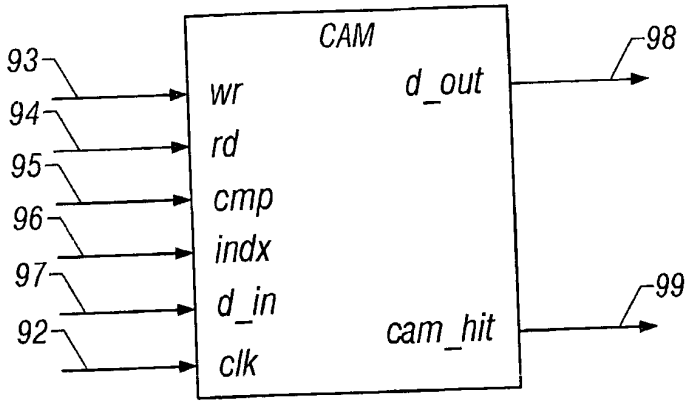

CAM
93
94
95
96
97
92
wr
rd
cmp
indx
d_in
clk
d_out
cam_hit
98
99

FIG. 5

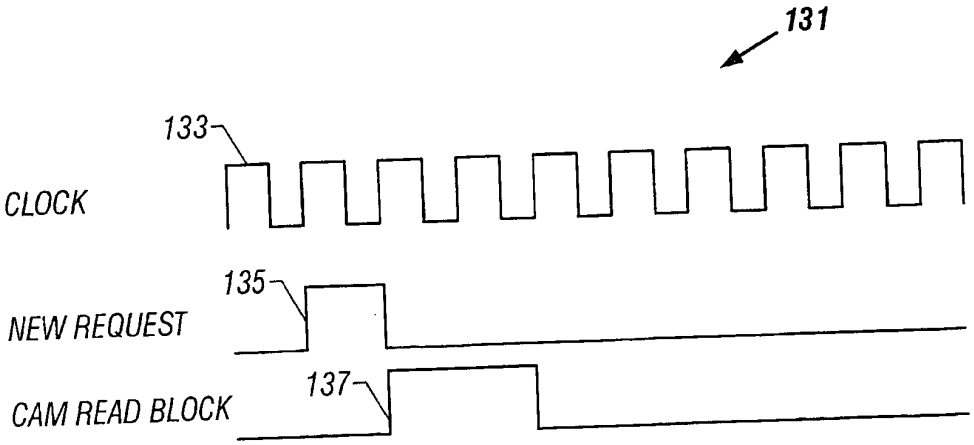

131
133
CLOCK
135
NEW REQUEST
137
CAM READ BLOCK

FIG. 6

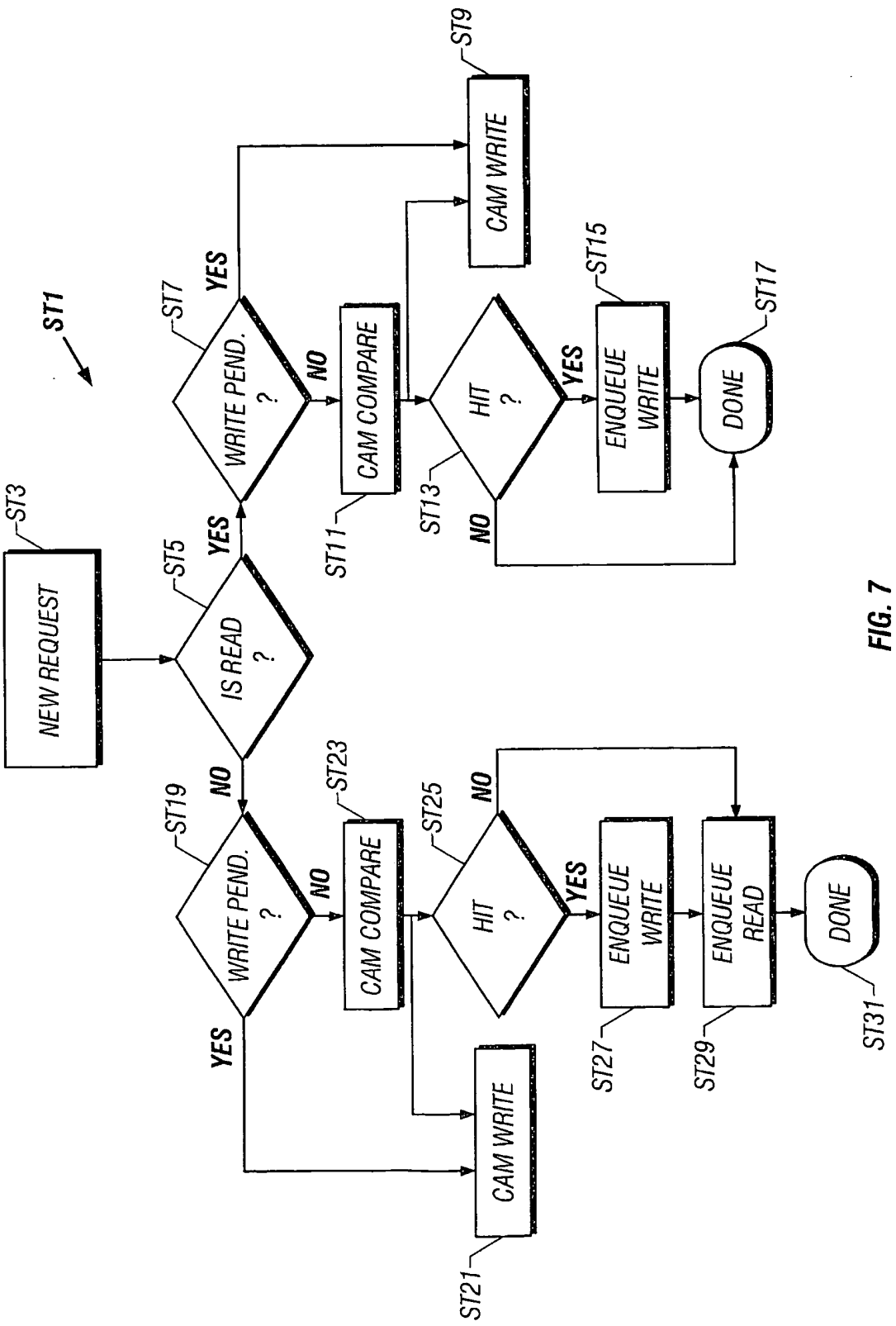

ST1
ST3
NEW REQUEST
ST5
IS READ ?
YES
NO
ST7
WRITE PEND. ?
YES
NO
ST9
CAM WRITE
ST11
CAM COMPARE
ST13
HIT ?
YES
NO
ST15
ENQUEUE WRITE
ST17
DONE
ST19
WRITE PEND. ?
YES
NO
ST21
CAM WRITE
ST23
CAM COMPARE
ST25
HIT ?
YES
NO
ST27
ENQUEUE WRITE
ST29
ENQUEUE READ
ST31
DONE

FIG. 7

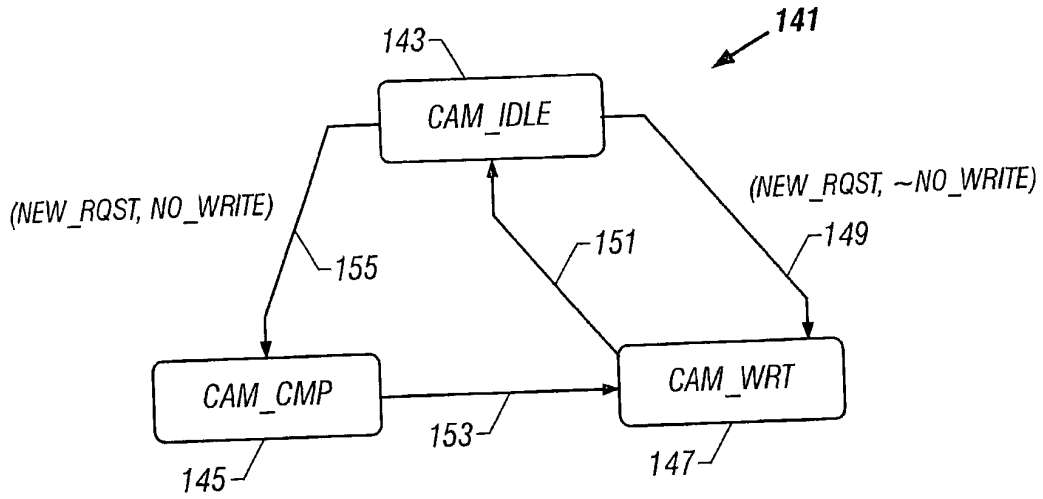

141
143
CAM_IDLE
(NEW_RQST, NO_WRITE)
(NEW_RQST, ~NO_WRITE)
155
151
149
CAM_CMP
CAM_WRT
153
145
147

FIG. 8

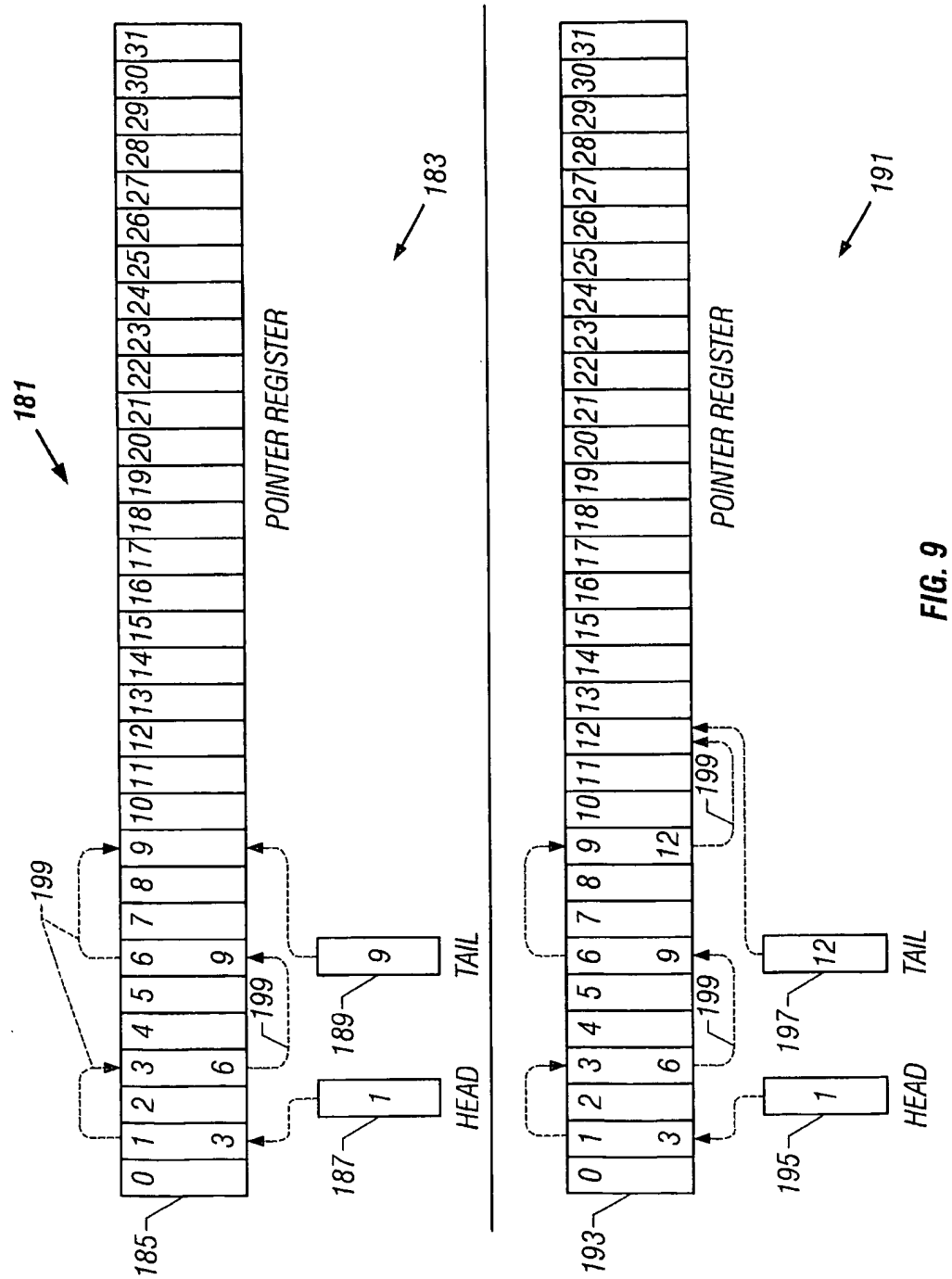

181
183
191
POINTER REGISTER
POINTER REGISTER
HEAD
TAIL
HEAD
TAIL
185
187
189
193
195
197
199

FIG. 9

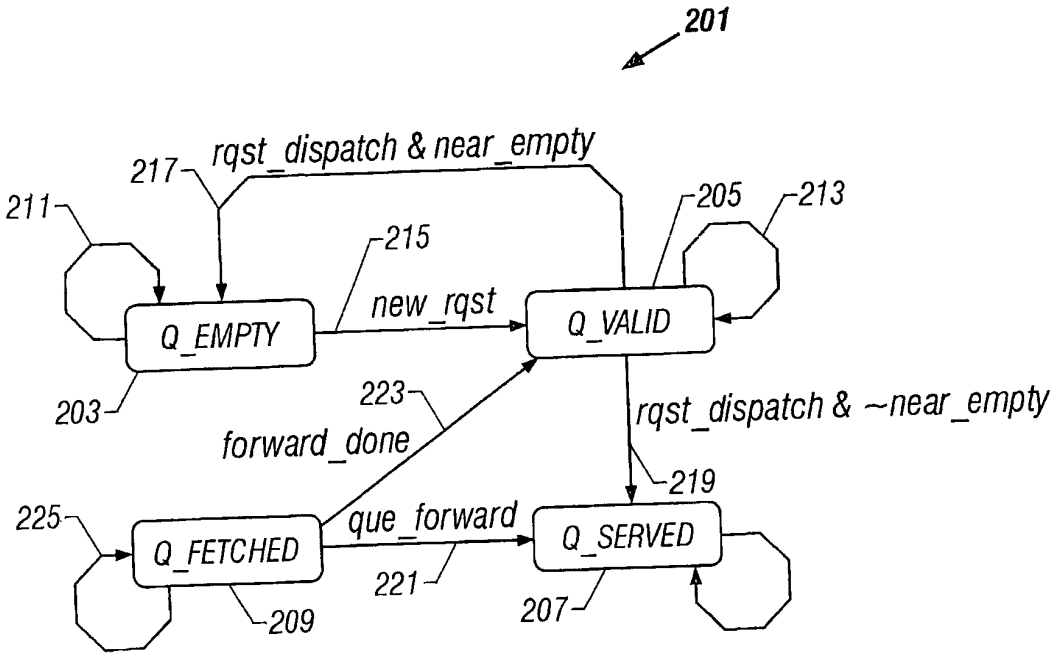

201
rqst_dispatch & near_empty
217
211
205
213
215
new_rqst
Q_EMPTY
Q_VALID
203
223
forward_done
rqst_dispatch & ~near_empty
219
225
que_forward
Q_FETCHED
Q_SERVED
221
209
207

FIG. 10